FACHHOCHSCHULE
TECHNIKUM WIEN

Fachbereich: Embedded Systems

Themengebiet: Computerarchitektur

FSM Implementierung

Version 2.2, August 2007

Peter Balog

Inhaltsverzeichnis

0. Übersicht

0.1. Lehrziele

Ziel des Studienbriefes *FSM Implementierung* liegt im Aufbau eines Verständnisses für die Spezifikation von sequentiellen Vorgängen und deren Implementierung mit geeigneten Modellen. Aufbauend auf den im Studienbrief *Sequentielle Logiksysteme* vorgestellten prinzipiellen Konzepten versucht der weiterführende Studienbrief den Zusammenhang von Spezifikation, Hardwaremodell und Implementierung von sequentiellen digitalen Systemen herzustellen.

0.2. Lehrinhalt

Anhand von Beispielen werden die Zustandskodierung und die FSM-Synthese für unterschiedliche Varianten des FSM-Modells erklärt.

0.3. Anmerkungen

Die Studienbriefe *Sequentielle Logiksysteme* und *FSM Implementierung* führen in die sequentiellen Systeme ein, wobei der Schwerpunkt bei den synchronen FSMs liegt. Die teilweise in die *Elektronik* hineinführenden Themen „Metastabilität“, „Hazards“, „hazardfreie“ Modelle, die „Gray“-Codierung, externe Taktsynchronisierung, synchrone Datenpfade (Pipelines) sowie asynchrone FSM-Modelle werden hier nicht behandelt.

1. FSM Implementierungen

Anhand einer verbalen Beschreibung eines einfachen sequentiellen Prozesses werden die STDs sowohl für Mealy- als auch für Moore-FSMs entworfen. Es wird gezeigt wie die kombinatorischen Logikfunktionen (*next state logic*, *output logic*) aus der STD-Spezifikation abgeleitet und schließlich synthetisiert (Synthese von kombinatorischen Funktionen !) werden können. Im Zuge des Entwurfsprozesses werden auch unterschiedliche Methoden zur Zustandskodierung gezeigt.

1.1. Allgemeines zur Zustandskodierung

Die Zustandskodierung ist die Kunst den i.A. symbolisch definierten Zuständen Bitmuster zuzuordnen, die dann im Zustandsregister gespeichert werden können. Die Art und Weise wie dies erfolgt ist in der Praxis sehr oft wirklich eine Kunst, da der Aufwand für die kombinatorischen Logikfunktionen stark von der Zustandskodierung abhängt. Meistens werden jedoch standardisierte Kodierungsverfahren verwendet:

- ***binary*** Zustandskodierung; benötigt minimale Anzahl von Flip-Flops
- ***1-hot*** Zustandskodierung; benötigt ein Flip-Flop je Zustand (maximale Anzahl)
- ***arbitrary*** Zustandskodierung; nützt die Eigenschaften der Anwendung aus; z.B. wenn interne Zustände gleich als Ausgangwert verwendet werden können (z.B. Zähler)

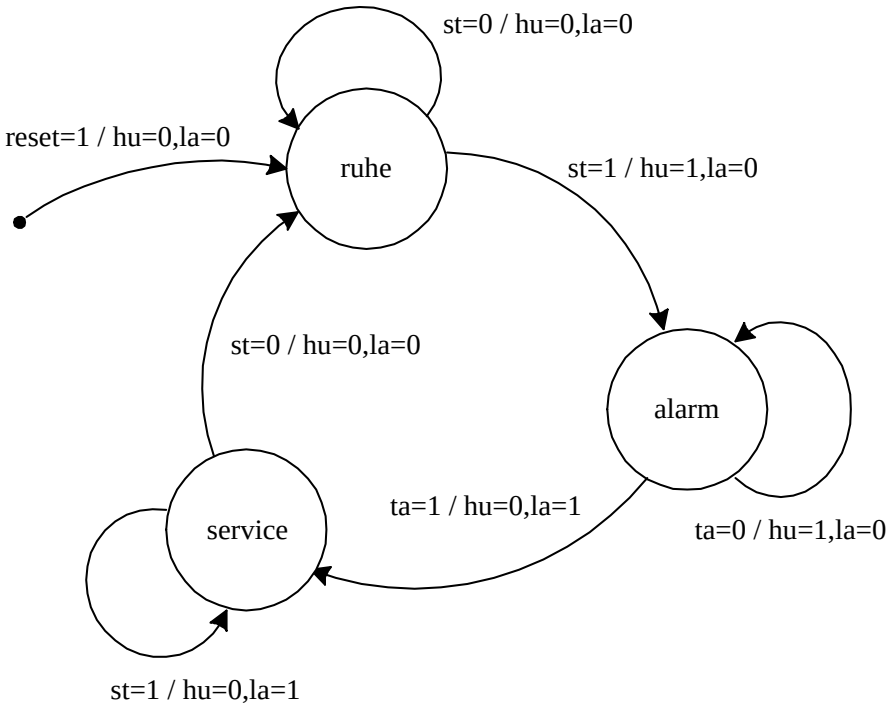
1.2. Aufgabenstellung

In einer Anlage wird eine Störung durch ein Signal (**st=1)** angezeigt. Die Anlage soll damit vom Normalzustand (**ruhe)** in den Alarmzustand (**alarm)** übergehen, in dem eine Hupe (**hu=1)** aktiviert wird, die einen Techniker herbeirufen soll. Dieser kann das aufdringliche Gehupe mit einer Taste (**ta=1)** abstellen. Liegt die Störung noch an, dann geht die Anlage in den Servicezustand (**service)**, der mit einer Lampe (**la=1**) signalisiert wird. Ist die Störung behoben, dann soll die Anlage wieder normal arbeiten.

2. Mealy-FSM mit binärer Zustandskodierung

2.1. Spezifikation mittels STD

- Die (symbolischen) Zustandsnamen (**ruhe**, **alarm**, **service**) benennen die Bubbles.
- Die Eingangsbedingungen und die Ausgänge werden zu den Transitionen notiert.
- Ein aktiver Reset (reset=1) führt aus allen Zuständen in den Zustand **ruhe**.



2.2. Binäre Zustandskodierung

Im Zuge der Zustandskodierung wird jedem i.A. symbolischen Zustandsnamen ein Bitmuster zugeordnet, welches im Zustandsregister gespeichert werden kann. Bei der **binären Zustandskodierung** geht man wie folgt vor:

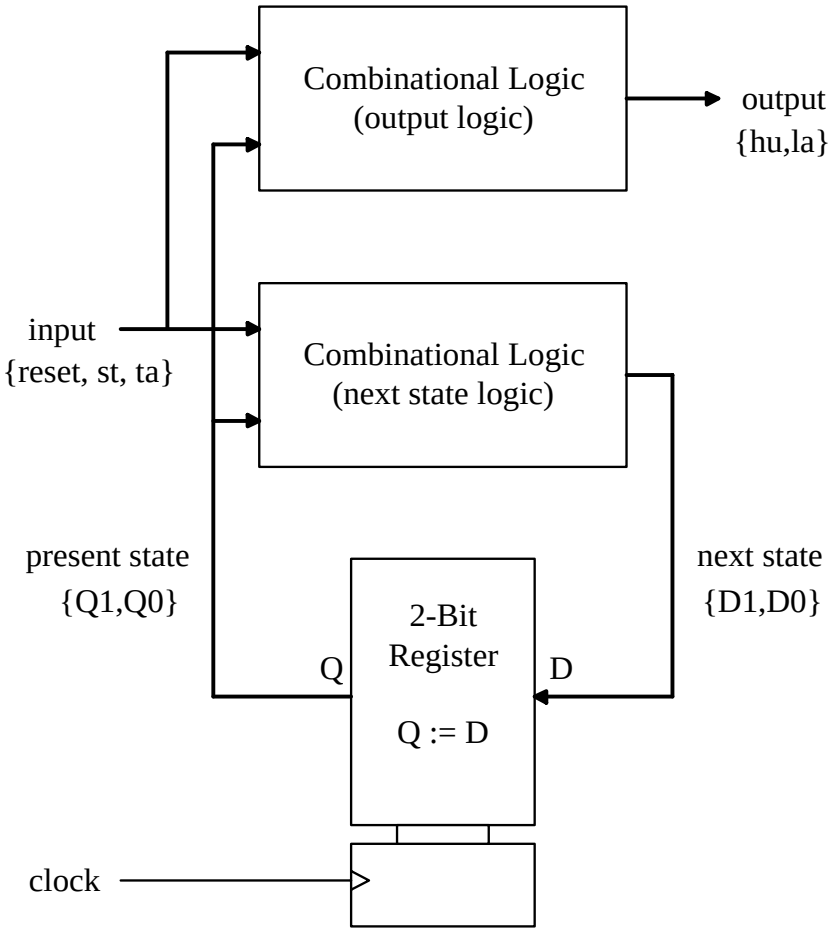
- Jedem symbolischen Zustandsnamen wird eine eindeutige Zustandsnummer (i.A. beginnend bei 0) zugeordnet.
- Die binäre Darstellung einer Zustandsnummer ist dann das entsprechende Zustandswort des Zustandsregisters.
- Die Anzahl der Zustände definiert die Breite des Zustandswortes (Breite des Registers bzw. Anzahl der Flip-Flops). Es gilt:

  Registerbreite = nächst_größeres_Ganzes (ld (Anzahl der Zustände))
- Da die Bitbreite des Registers eine ganze Zahl sein muss ergibt sich die Zahl der kodierbaren Zustände (i.A größer als die Anzahl der Zustände) zu:

  $Anzahl\ der\ darstellbaren\ Zustände = 2^{Registerbreite}$
- Die nicht verwendeten Zustandsworte (ungenutzte Zustände, illegale Zustände) müssen i.A. bei der Synthese berücksichtigt werden, damit die FSM (für alle Fälle) ein deterministisches Verhalten hat.

Zustandsname	Zustandsnummer	Zustandskodierung	
		Q1	Q0
ruhe	0	0	0
alarm	1	0	1
service	2	1	0
ungenutzt	3	1	1

2.3. Das Hardware-Modell



Anmerkung:

Das STD beschreibt das funktionale Transitionsverhalten eines sequentiellen Prozesses. Es definiert jedoch in keinster Weise, zu welchem konkreten Zeitpunkt ein Übergang stattfindet. Dies ist durch das Hardware-Modell festgelegt.

2.4. Transformation des STD in eine Tabelle

Das STD beschreibt das Verhalten der FSM mit einem Graphen. Dieser Graph kann in eine Tabelle umgesetzt werden, welche die notwendige Information zur Synthese der **next state**- und der **output**-Funktion enthält. Die Struktur der Tabelle ist die folgende:

- present state | inputs | next state | output
- Aus dem **present state** und den **inputs** wird der **next state** berechnet.
- Aus dem **present state** und den **inputs** werden die **outputs** berechnet.

Kommentar	present state		inputs			next state		outputs	
	Q1	Q0	reset	st	ta	D1	D0	hu	la
für alle Zustände	x	x	1(1)	x	x	0	0	0	0
für Zustand **ruhe**	0	0	0	0	x	0	0	0	0
	0	0	0	1	x	0	1	1	0
für **alarm**	0	1	0	x	0	0	1	1	0
	0	1	0	x	1	1	0	0	1
für **service**	1	0	0	0	x	0	0	0	0
	1	0	0	1	x	1	0	0	1
für **ungenutzt**	1	1	0(2)	x	x	0	0	0(3)	0(3)

Anmerkungen:

(1) durch diese Zeile werden alle Reset-Bedingungen abgedeckt.

(2) hier könnte auch ein X stehen, jedoch ist dieser Fall bereits durch die 1. Zeile erfasst.

(3) hier hätte man die Möglichkeit hu=1 und la=1 zu setzen, um den ungenutzten (illegalen) Zustand am Ausgang „sichtbar“ zu machen.

2.5. Synthese der *next state logic*

Um die **next state** Funktion zu synthetisieren extrahiert man die relevanten Teile der STD-Tabelle um so die notwendige Wahrheitstabelle zu erhalten. Für die **next state** Funktion gilt:

- present state und inputs bilden die Eingänge
- next state bildet die Ausgänge

	Eingänge					Ausgänge	
	present state		inputs			next state	
Z.Nr.	Q1	Q0	reset	st	ta	D1	D0
rest[(1)]	x	x	1	x	x	0	0
0,1	0	0	0	0	x	0	0
2,3	0	0	0	1	x	0	1
8,10	0	1	0	x	0	0	1
9,11	0	1	0	x	1	1	0
16,17	1	0	0	0	x	0	0
18,19	1	0	0	1	x	1	0
24-27	1	1	0	x	x	0	0

(1) Z.Nr.: 4-7, 12-15, 20-23, 28-31

- Damit man das Umschreiben in eine geordnete Wahrheitstabelle erspart, werden die Zeilen bzw. Zeilenbereiche entsprechend beschriftet.
- Da die Funktionen für D1 und D0 nur wenige 1 enthalten ist ein KV-Diagramm nicht notwendig: D1={(m9,m11),(m18,m19)}, D0={(m2,m3),(m8,m10)}

$$D1 = \overline{Q1} \cdot Q0 \cdot \overline{reset} \cdot ta + Q1 \cdot \overline{Q0} \cdot \overline{reset} \cdot st$$

$$D0 = \overline{Q1} \cdot \overline{Q0} \cdot \overline{reset} \cdot st + \overline{Q1} \cdot Q0 \cdot \overline{reset} \cdot \overline{ta}$$

2.6. Synthese der *output logic*

Um die **output** Funktion zu synthetisieren extrahiert man die relevanten Teile der STD-Tabelle um so die notwendige Wahrheitstabelle zu erhalten. Für die **output** Funktion gilt:

- present state und inputs bilden die Eingänge
- outputs bilden die Ausgänge

	Eingänge					Ausgänge	
	present state		inputs			outputs	
Z.Nr.	Q1	Q0	reset	st	ta	hu	la
rest[(1)]	x	x	1	x	x	0	0
0,1	0	0	0	0	x	0	0
2,3	0	0	0	1	x	1	0
8,10	0	1	0	x	0	1	0
9,11	0	1	0	x	1	0	1
16,17	1	0	0	0	x	0	0
18,19	1	0	0	1	x	0	1
24-27	1	1	0	x	x	0	0

(1) Z.Nr.: 4-7, 12-15, 20-23, 28-31

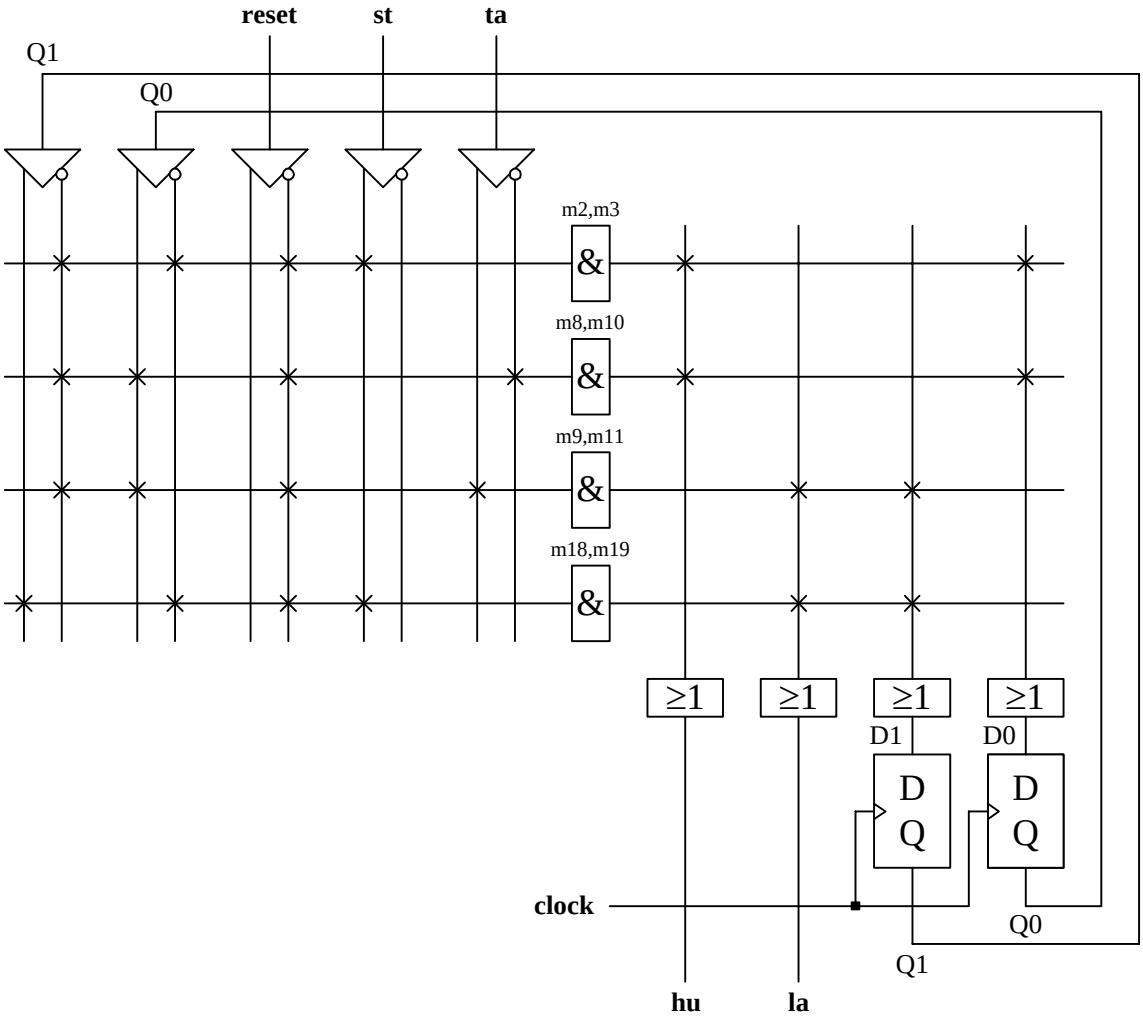
- Da die Funktionen für **hu** und **la** nur wenige 1 enthalten ist ein KV-Diagramm nicht notwendig: hu={(m2,m3),(m8,m10)}, la={(m9,m11),(m18,m19)}

$$hu = \overline{Q1} \cdot \overline{Q0} \cdot \overline{reset} \cdot st + \overline{Q1} \cdot Q0 \cdot \overline{reset} \cdot \overline{ta}$$

$$la = \overline{Q1} \cdot Q0 \cdot \overline{reset} \cdot ta + Q1 \cdot \overline{Q0} \cdot \overline{reset} \cdot st$$

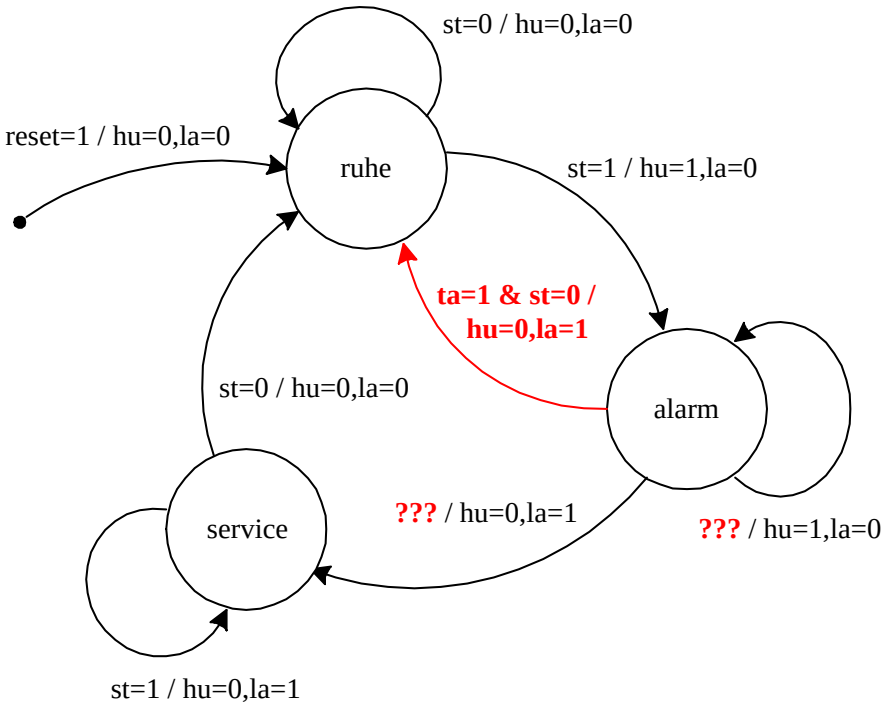
2.7. PLA-Implementierung

- Erweiterung der PLA-Struktur um Flip-Flops nach den Summenfunktionen.
- Flip-Flop-Ausgänge können in das UND-Feld rückgekoppelt werden.
- Die sogenannte *Registered PLA* Struktur ist eine Implementierungsarchitektur für (synchrone) FSMs.



3. Erweiterung der Spezifikation

Wenn der Servicetechniker die Taste drückt um das aufdringliche Gehupe in ein sanftes Glimmen der Lampe zu ändern und die Störung hat sich in der Zwischenzeit „selbst behoben“, dann soll **sofort** wieder der Normalzustand erreicht werden (siehe Übergang von **alarm** nach **ruhe**).



Aufgabe 1:

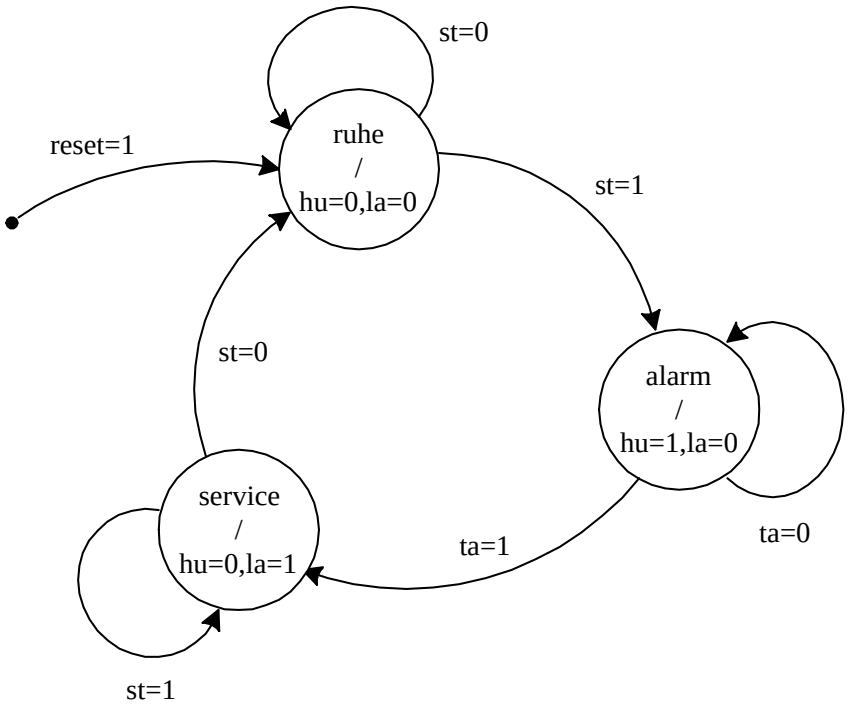
Ergänzen Sie obige Spezifikation derart, dass die STD-Spezifikation konsistent ist.

(Lösung siehe Seite 26)

4. Moore-FSM mit 1-hot Zustandskodierung

4.1. Spezifikation mittels STD

- Die (symbolischen) Zustandsnamen (**ruhe**, **alarm**, **service**) benennen die Bubbles.
- Die Eingangsbedingungen werden zu den Transitionen notiert.
- Die Ausgänge werden in die Bubbles plaziert, da sie nur mehr vom Zustand abhängen.
- Ein aktiver Reset (reset=1) führt aus allen Zuständen in den Zustand **ruhe**.



4.2. 1-hot Zustandskodierung

Im Zuge der Zustandskodierung wird jedem i.A. symbolischen Zustandsnamen ein Bitmuster zugeordnet, welches im Zustandsregister gespeichert werden kann. Bei der **1-hot Zustandskodierung** geht man wie folgt vor:

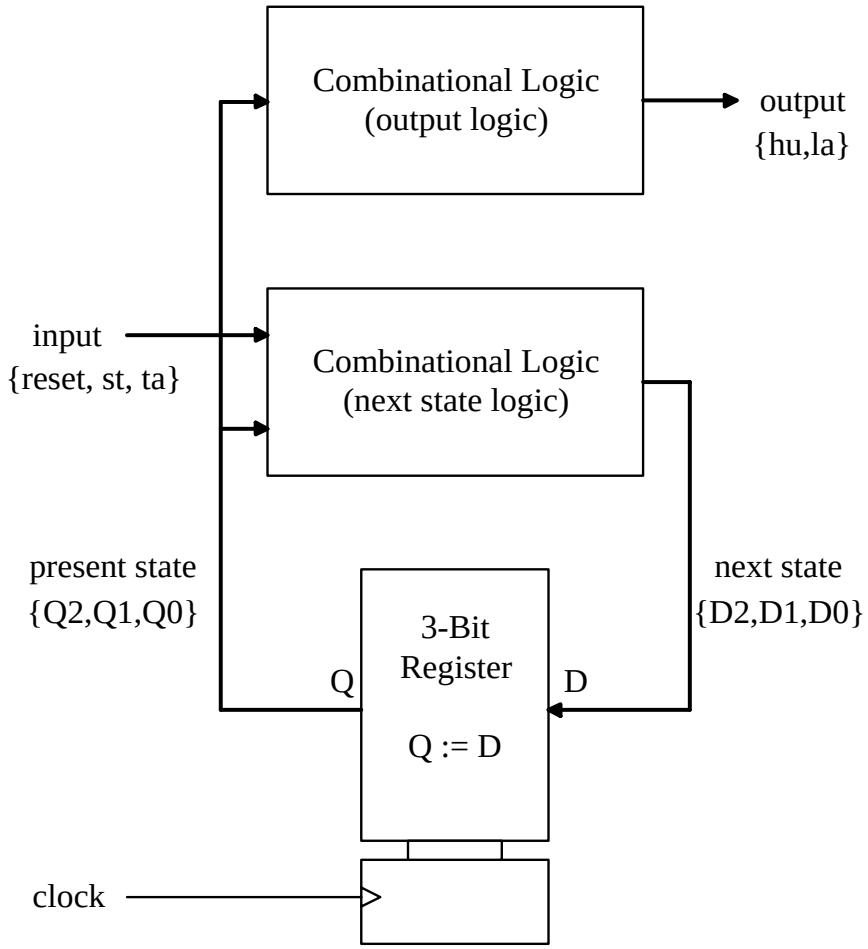
- Jedem symbolischen Zustandsnamen wird eine Bitposition im Zustandswort zugeordnet. Oder anders formuliert,- für jeden Zustand benötigt man ein Flip-Flop.
- Die Anzahl der Zustände definiert die Breite des Zustandswortes (Breite des Registers bzw. Anzahl der Flip-Flops). Es gilt:

  Registerbreite = Anzahl der Zustände
- Ein Zustandswort ist „dann und nur dann“ gültig, wenn es genau an einer Bitposition des Zustandswortes eine 1 aufweist.

Zustandsname	Zustandskodierung		
	Q2	Q1	Q0
ruhe	0	0	1
alarm	0	1	0
service	1	0	0

- Infolge der Definition gibt es keine ungenutzten (legalen) Zustandcodes. Dies bezieht sich jedoch nur auf die gültigen bzw. zulässigen Codewörter.
- Implementierungsbedingt gibt es jede Menge illegale Codewörter. Diese erzeugen bei der Synthese jedoch keinen „Code“ (Hardware), wenn sie in das **0-Wort** übergeführt werden (der Folgezustand aus jedem illegalen Zustand ist das 0-Wort).
- Sinnvollerweise wechselt man aus dem (illegalen) **0-Wort** unbedingt in einen gültigen Zustand. Dies wird jedoch in der STD-Spezifikation nicht eingetragen.

4.3. Das Hardware-Modell



Anmerkung:

Die Ausgänge (hu, la) sind nur mehr Funktion des aktuellen Zustands (present state). Vergleich dazu die Defintion der Moore-Ausgänge im Studienbrief 5.

4.4. Transformation des STD in eine Tabelle

Das STD beschreibt das Verhalten der FSM mit einem Graphen. Dieser Graph kann in eine Tabelle umgesetzt werden, welche die notwendige Information zur Synthese der **next state**- und der **output**-Funktion enthält. Die Struktur der Tabelle ist die folgende:

- present state | inputs | next state | output
- Aus dem **present state** und den **inputs** wird der **next state** berechnet.
- Aus dem **present state** werden die **outputs** berechnet.

Kommentar	present state			inputs			next state			outputs	
	Q2	Q1	Q0	reset	st	ta	D2	D1	D0	hu	la
für alle Zustände	x	x	x	**1**	x	x	0	0	1	0	0
für Zustand **ruhe**	0	0	1	0	0	x	0	0	1	0	0
	0	0	1	0	1	x	0	1	0	0	0
für **alarm**	0	1	0	0	x	0	0	1	0	1	0
	0	1	0	0	x	1	1	0	0	1	0
für **service**	1	0	0	0	0	x	0	0	1	0	1
	1	0	0	0	1	x	1	0	0	0	1
für **0-Wort**	**0**	**0**	**0**	0	x	x	0	0	1	0	0
für alle anderen(1)	y	y	y	0	x	x	**0**	**0**	**0**	0	0

Anmerkungen:

(1) für alle illegalen Zustandsworte mit Ausnahme des 0-Wortes.

4.5. Synthese der *next state logic*

Um die **next state** Funktion zu synthetisieren extrahiert man die relevanten Teile der STD-Tabelle um so die notwendige Wahrheitstabelle zu erhalten. Für die **next state** Funktion gilt:

- present state und inputs bilden die Eingänge
- next state bildet die Ausgänge

	Eingänge						Ausgänge		
	present state			inputs			next state		
Zeilennummern	Q2	Q1	Q0	reset	st	ta	D2	D1	D0
global reset(1)	x	x	x	1	x	x	0	0	1
8,9	0	0	1	0	0	x	0	0	1
10,11	0	0	1	0	1	x	0	1	0
16,18	0	1	0	0	x	0	0	1	0
17,19	0	1	0	0	x	1	1	0	0
32,33	1	0	0	0	0	x	0	0	1
34,35	1	0	0	0	1	x	1	0	0
0-3	**0**	**0**	**0**	0	x	x	0	0	1
rest(2)	y	y	y	0	x	x	**0**	**0**	**0**

(1) 4-7,12-15,20-23,28-31,36-39,44-47,52-55,60-63 (32er-Block für reset=1)
(2) die Minterme sind für die Funktion(en) irrelevant, da sie zum OFF-Set gehören.

- Die Funktionen D2 und D1 lassen sich direkt aus der Wahrheitstabelle ablesen:

$$D2 = \overline{Q2} \cdot Q1 \cdot \overline{Q0} \cdot \overline{reset} \cdot ta + Q2 \cdot \overline{Q1} \cdot \overline{Q0} \cdot \overline{reset} \cdot st$$

$$D1 = \overline{Q2} \cdot Q1 \cdot \overline{Q0} \cdot \overline{reset} \cdot \overline{ta} + \overline{Q2} \cdot \overline{Q1} \cdot Q0 \cdot \overline{reset} \cdot st$$

- Für die Funktion D0 benötigen wir ein KV-Diagramm, da diese Funktion sehr oft 1 am Ausgang liefert:

reset,st,ta \ Q2,Q1,Q0	000	001	011	010	110	111	101	100
000	**1** 0	**1** 8	24	16	48	56	40	**1** 32
001	**1** 1	**1** 9	25	17	49	57	41	**1** 33
011	**1** 3	11	27	19	51	59	43	35
010	**1** 2	10	26	18	50	58	42	34
110	**1** 6	**1** 14	**1** 30	**1** 22	**1** 54	**1** 62	**1** 46	**1** 38
111	**1** 7	**1** 15	**1** 31	**1** 23	**1** 55	**1** 63	**1** 47	**1** 39
101	**1** 5	**1** 13	**1** 29	**1** 21	**1** 53	**1** 61	**1** 45	**1** 37
100	**1** 4	**1** 12	**1** 28	**1** 20	**1** 52	**1** 60	**1** 44	**1** 36

- Primimplikanten:

  p1={mi: reset=1} (32er-Block der unteren Hälfte)

  p2={m0,m1,m2,m3,m4,m5,m6,m7} (8er Reihe ganz links)

  p3={m0,m1,m8,m9,m4,m5,m12,m13} (4 links oben, 4 links unten)

  p4={m0,m1,m32,m33,m4,m5,m36,m37} (4 mal 2er-Blöcke in den Ecken)

- Die Logikfunktion in DNFmin für D0 lautet:

$$D0 = reset + \overline{Q2} \cdot \overline{Q1} \cdot \overline{Q0} + \overline{Q2} \cdot \overline{Q1} \cdot \overline{st} + \overline{Q1} \cdot \overline{Q0} \cdot \overline{st}$$

4.6. Synthese der *output logic*

Um die **output** Funktion zu synthetisieren extrahiert man die relevanten Teile der STD-Tabelle um so die notwendige Wahrheitstabelle zu erhalten. Für die **output** Funktion gilt:

- present state bildet die Eingänge
- outputs bilden die Ausgänge

	Eingänge			Ausgänge	
	present state			outputs	
Zeilennummern	Q2	Q1	Q0	hu	la
1	0	0	1	0	0
2	0	1	0	1	0
4	1	0	0	0	1
0	**0**	**0**	**0**	0	0
rest(1)	y	y	y	0	0

(1) die Minterme sind für die Funktion(en) irrelevant, da sie zum OFF-Set gehören.

- Da die Funktionen für **hu** und **la** nur jeweils einen 1 in der Ausgangsspalte aufweisen, kann die Funktion direkt angeschrieben werden:

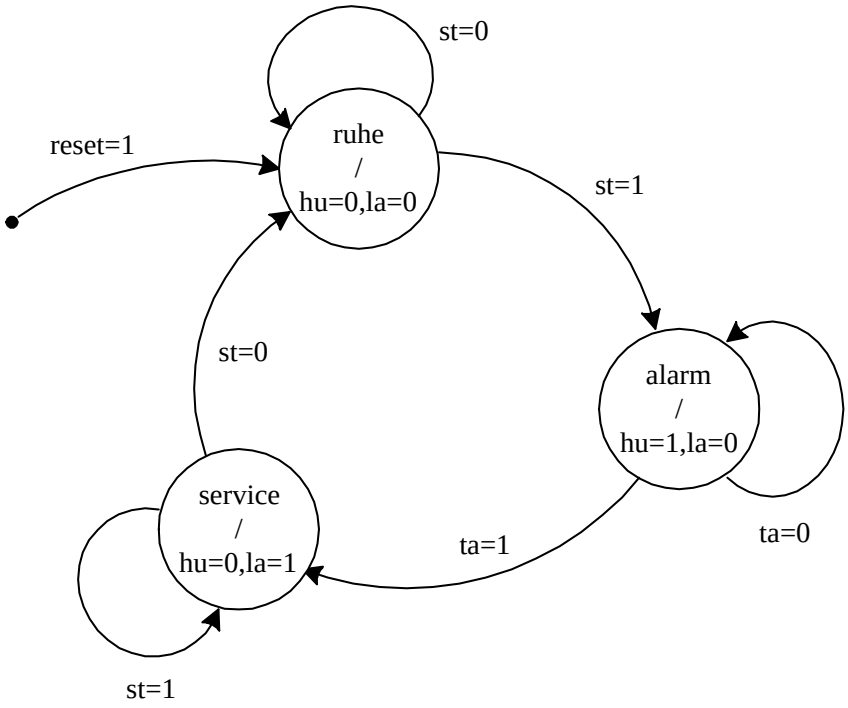
$$hu = \overline{Q2} \cdot Q1 \cdot \overline{Q0}$$

$$la = Q2 \cdot \overline{Q1} \cdot \overline{Q0}$$

5. Moore-FSM mit *arbitrary* Zustandskodierung

5.1. Spezifikation mittels STD

- Die STD-Spezifikation ist ident zum Kapitel 4.



5.2. Die *arbitrary* Zustandskodierung

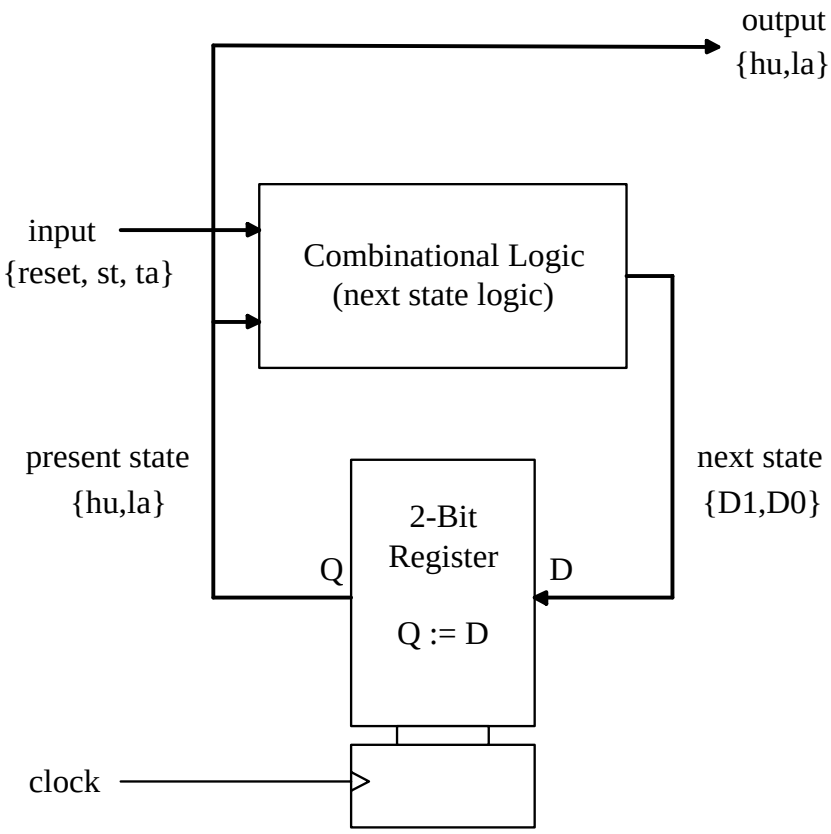
Die Idee der arbitrary Zustandskodierung geht davon aus, dass die Ausgangswerte in unterschiedlichen Zuständen die Zustände eindeutig identifizieren. Das klassische Beispiel ist ein binärer Zähler: der Zählstand ist sowohl Ausgang als auch Zustand.

- Die drei Zustände **ruhe**, **alarm** und **service** werden durch die Ausgänge **hu** und **la** eindeutig identifiziert.

Zustandsname	Zustandskodierung	
	Q1=**hu**	Q0=**la**
ruhe	0	0
alarm	1	0
service	0	1
ungenutzt	1	1

- Die vierte Möglichkeit (hu=1 und la=1) wird nicht ausgenützt. Dieser ungenutzte oder illegale Zustand wird durch die Ausgänge somit ebenfalls eindeutig identifiziert.

5.3. Das Hardware-Modell



Anmerkung:

Das Hardware-Modell zeigt recht eindrucksvoll die Motivation der *arbitrary* Kodierung. Die Ausgangslogik wird durch diese Art der Zustandskodierung i.A. sehr einfach; oder sie entfällt komplett wie in diesem Beispiel.

5.4. Transformation des STD in eine Tabelle

Das STD beschreibt das Verhalten der FSM mit einem Graphen. Dieser Graph kann in eine Tabelle umgesetzt werden, welche die notwendige Information zur Synthese der **next state**- und der **output**-Funktion enthält. Die Struktur der Tabelle ist für das folgende Beispiel die folgende:

- present state = outputs | inputs | next state
- Aus dem **present state** und den **inputs** wird der **next state** berechnet.
- Die **outputs** sind ident dem **present state**.

Kommentar	present state = outputs		inputs			next state	
	hu	la	reset	st	ta	D1	D0
für alle Zustände	x	x	1	x	x	0	0
für Zustand **ruhe**	0	0	0	0	x	0	0
	0	0	0	1	x	1	0
für **alarm**	1	0	0	x	0	1	0
	1	0	0	x	1	0	1
für **service**	0	1	0	0	x	0	0
	0	1	0	1	x	0	1
für **ungenutzt**	1	1	0	x	x	0	0

Aufgabe 2:

Synthetisieren Sie ***next state logic*** (D1, D0) in DNFmin. (Lösung siehe Seite 27)

6. Lehrzielorientierte Fragen

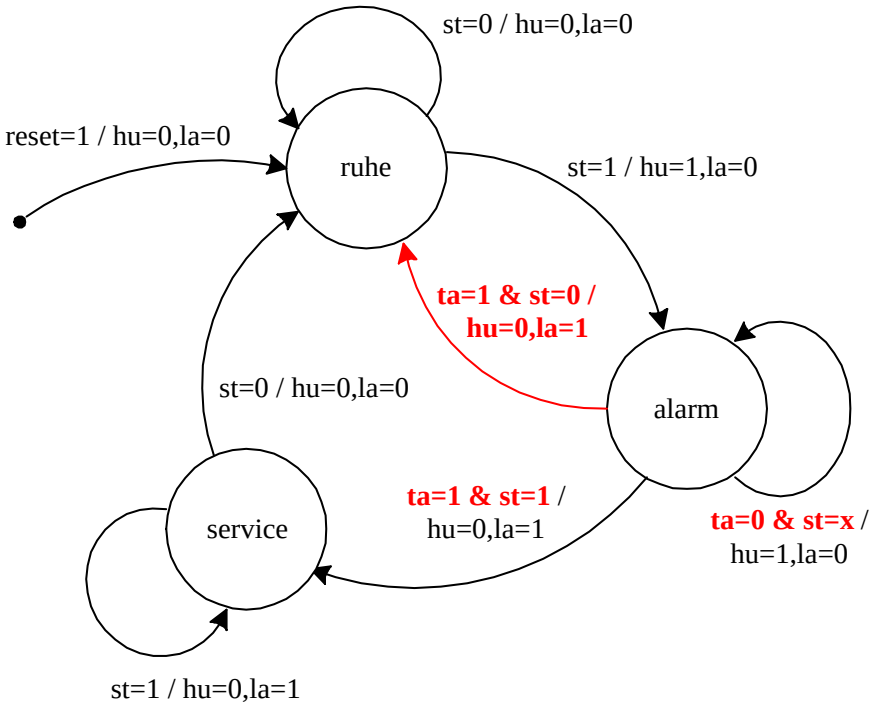
1. Was versteht man unter Zustandskodierung?
2. Welche Arten von Zustandskodierung sind Ihnen bekannt?
3. Welche Eigenschaften haben die unterschiedlichen Zustandskodierungsverfahren?
4. Kann sich die Spezifikation einer FSM durch die Zustandskodierung ändern?
5. Was versteht man unter einer konsistenten FSM-Spezifikation?
6. Kann die Konsistenz durch die Zustandskodierung beeinflusst werden?

6.1. Antworten auf die lehrzielorientierten Fragen

1. Durch die Zustandskodierung wird festgelegt, wie die Zustände mit ihren i.A. symbolischen Zustandsnamen auf das Zustandsregister abgebildet werden.
2. Binär, 1-Hot, Arbitrary
3. Bei binärer Zustandkodierung benötigt man die minimalste Anzahl von Bits im Zustandsregister (minimale Anzahl von Flip-Flops). Bei 1-Hot Kodierung wird jedem Zustand ein Bit im Register (ein Flip Flop) zugeordnet. Bei Arbitrary Kodierung wird versucht, Ausgangswerte direkt auf Zustandswerte abzubilden.
4. Ja, da i.A. nicht alle Bitmuster des Zustandsregisters nach der Zustandskodierung ausgenutzt werden. Das Verhalten der FSM bzgl. dieser ungenutzten bzw. illegealen Zustände kann/muss in der Spezifikation berücksichtigt werden.
5. Eindeutige Zustände (sowohl Namen als auch Kodierung), für jeden Zustand sind alle (relevanten) Eingangsbedingungen eindeutig erfasst.
6. Ja, da die Anzahl der Zustände durch die Zustandskodierung erhöht werden kann.

7. Lösungen

Lösung zu Aufgabe 1:



Lösung zu Aufgabe 2:

Kommentar	present state = outputs		inputs			next state	
	hu	la	reset	st	ta	D1	D0
für alle Zustände	x	x	1	x	x	0	0
für Zustand **ruhe**	0	0	0	0	x	0	0
	0	0	0	1	x	1	0
für **alarm**	1	0	0	x	0	1	0
	1	0	0	x	1	0	1
für **service**	0	1	0	0	x	0	0
	0	1	0	1	x	0	1
für **ungenutzt**	1	1	0	x	x	0	0

$$D1 = \overline{hu} \cdot \overline{la} \cdot \overline{reset} \cdot st + hu \cdot \overline{la} \cdot \overline{reset} \cdot \overline{ta}$$

$$D0 = hu \cdot \overline{la} \cdot \overline{reset} \cdot ta + \overline{hu} \cdot la \cdot \overline{reset} \cdot st$$